From Fix to Flaw: Understanding and Revealing Incomplete Patches for Link Following Vulnerabilities

Bocheng Xiang
Fudan University
Shanghai, China
bcxiang23@m.fudan.edu.cn

Yuan Zhang*
Fudan University
Shanghai, China
yuanxzhang@fudan.edu.cn

Hao Huang
Fudan University
Shanghai, China
huangh21@m.fudan.edu.cn

Youkun Shi
Fudan University
Shanghai, China
21110240048@m.fudan.edu.cn

Abstract

Link Following vulnerabilities (LFVulns) remain a persistent and high-impact threat in modern operating systems, enabling attackers to redirect privileged file operations to unintended targets via symbolic links. While substantial efforts have been devoted to detecting, exploiting, and mitigating LFVulns, considerably less attention has been paid to the correctness and robustness of real-world LFVuln patches themselves. In practice, the complexity of file operation semantics not only makes LFVulns difficult to prevent but also renders their remediation error-prone, leading to patches that appear correct yet remain vulnerable to bypass.

In this paper, we present the first systematic empirical study of incomplete LFVuln patches. To support this study, we design *LinkDiff*, a binary differential analysis framework for analyzing LFVuln patches in closed-source software by combining semantics-guided diff selection with LLM-assisted reasoning over file-operation call chains. Using *LinkDiff*, we analyze 60 historical incomplete LFVuln patches. Based on an in-depth root-cause analysis, we derive a fine-grained taxonomy comprising 9 distinct types grouped into three high-level categories: under-specified path validation, flawed file-metadata assumptions, and insufficient privilege isolation.

We further operationalize the taxonomy by integrating its findings into *LinkDiff*, enabling automated identification and classification of incomplete LFVuln patches in practice. Applying the enhanced *LinkDiff* to 76 latest LFVuln patches from 62 closed-source applications, it flags 63 as incomplete. Manual validation shows that 7 of them are false positives, while the remaining 56 are indeed incomplete, indicating that incomplete LFVuln patches remain prevalent in today's software ecosystem. Among these, 26 can directly lead to new LFVulns, resulting in 19 CVEs and $28,100 in bug bounties through responsible disclosure.

*Corresponding author

CCS '26, The Hague, Netherlands.

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832682

CCS Concepts

• **Security and privacy → Operating systems security**.

Keywords

File System, Software Security, Patch Analysis

ACM Reference Format:
Bocheng Xiang, Yuan Zhang, Hao Huang, and Youkun Shi. 2026. From Fix to Flaw: Understanding and Revealing Incomplete Patches for Link Following Vulnerabilities. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832682

1 Introduction

File systems are a foundational component of modern operating systems, and link mechanisms such as symbolic links are widely used to support flexible file management. However, the indirection introduced by these mechanisms also enables Link Following vulnerabilities (LFVulns), where attackers abuse links to redirect privileged file operations to unintended sensitive targets. Such vulnerabilities can lead to severe security consequences, including denial of service, local privilege escalation, and sensitive information disclosure. In practice, the complexity of file operation semantics not only makes LFVulns difficult to prevent during development but also renders their remediation error-prone, and as a result, hundreds of LFVuln patches have been bypassed and re-fixed over time, leading to recurring vulnerabilities caused by incomplete patches.

In real-world applications, LFVulns are patched by developers with diverse development practices and security expertise, who often adopt different perspectives and strategies when implementing fixes. While some patches fully eliminate the vulnerability, many others are incomplete and fail to comprehensively address the underlying issues. Whether such incomplete patches introduce security risks, and how they may eventually lead to new LFVulns, remain open questions. Motivated by these concerns, this work conducts a comprehensive empirical analysis of incomplete LFVuln patches by designing a differential analysis framework to systematically locate and examine such patches, to understand and reveal their characteristics and the latent security risks they introduce, intending to inform more robust remediation practices and improve the community's understanding of LFVuln patching pitfalls.

Existing research on LFVulns can be broadly categorized into two main lines of work: 1) *LFVuln detection and exploitation.* These studies [1–6] focus on identifying and exploiting LFVulns through static analysis or hybrid static–dynamic techniques. For example, PathSentinel [2] employs static analysis combined with access control policy modeling to detect LFVulns in Android systems, while LinkZard [1] leverages fuzzing-based dynamic exploration and subgraph matching to uncover LFVulns in real-world applications. 2) *LFVuln Mitigation and defense.* This line of work [7–9] aims to prevent LFVulns at runtime by dynamically monitoring file path resolution and access behaviors, either in the user space or kernel space, thereby enforcing real-time protection. However, the above studies largely overlook the security risks that may arise during the patching and remediation of LFVulns. To the best of our knowledge, no prior work has examined the security implications of LFVuln patches themselves, let alone the problem of identifying incomplete LFVuln patches or developing a systematic understanding of their characteristics and root causes.

To support our empirical analysis, we first design and develop a binary differential analysis framework, named *LinkDiff*, to systematically identify and analyze LFVuln patches. It adopts a two-stage design. In the first stage, it constructs a difference database and then efficiently prunes global code changes by building bottom-up file-operation call chains, motivated by the critical observation that LFVuln patches tend to manifest along such call chains, yielding a focused set of candidate patches. In the second stage, *LinkDiff* leverages an LLM-driven agent [10] equipped with multiple auxiliary functions for binary program analysis, enabling systematic reasoning over patch semantics to identify candidate LFVuln patches. Leveraging *LinkDiff*, we conduct an empirical analysis on a set of 60 historical incomplete LFVuln patches. Through in-depth root-cause analysis, we systematically examine how these patches fail and the security risks they introduce. Based on this analysis, we present the first fine-grained taxonomy of incomplete LFVuln patches, categorizing them into 9 distinct types, which further fall into three high-level categories: Under-Specified Path Validation, Flawed File-Metadata Assumptions, and Insufficient Privilege Isolation.

To further assist developers in evaluating LFVuln patches during development, we extend *LinkDiff* with the capability to identify and categorize incomplete patches by incorporating the taxonomy-derived patterns and characteristic features uncovered in our empirical study. Specifically, inspired by retrieval-augmented generation (RAG [11]), we build an external knowledge base [12] for the identified patch types, containing their key characteristics, real-world examples, and representative patch snippets. This knowledge base helps *LinkDiff* ground its semantic reasoning in domain-specific knowledge for more accurate identification and fine-grained classification of incomplete patches. On the ground-truth dataset of 60 historical incomplete LFVuln patches, *LinkDiff* correctly assigns the exact subtype to 49 cases, achieving 94.23% taxonomy-aware precision and 85.96% taxonomy-aware recall. We further evaluate the enhanced *LinkDiff* on 76 latest LFVuln patches across 25 vendors. It flags 63 as incomplete; manual validation shows that 7 are false positives, while the remaining 56 are indeed incomplete, demonstrating the prevalence of incomplete LFVuln remediation in contemporary software. Among them, 26 can directly lead to new LFVulns. We responsibly reported these cases, together with corresponding exploitation details, to affected vendors, including Microsoft, Intel, and Adobe, resulting in 19 assigned CVEs and $28,100 in bug bounties.

In summary, we make the following contributions:

- We design and implement *LinkDiff*, a binary differential analysis framework for systematically locating, identifying, and analyzing LFVuln patches in real-world software (§4).
- Using *LinkDiff*, we conduct the first systematic empirical study of incomplete LFVuln patches, and provide a fine-grained characterization of their recurring patterns, underlying causes of incompleteness. Concretely, we derive a taxonomy of 9 distinct types and analyze the security risks they introduce (§5).
- We operationalize our empirical findings by enhancing *LinkDiff* to automatically identify and classify incomplete LFVuln patches. On 60 historical incomplete patches, *LinkDiff* correctly assigns the exact subtype to 49 cases, achieving 94.23% taxonomy-aware precision and 85.96% taxonomy-aware recall (§6.2).
- We then analyze 76 latest released LFVuln patches from 62 closed-source applications using *LinkDiff* and find that 56 of them remain incomplete, demonstrating that incomplete LFVuln remediation remains prevalent in today's software ecosystem. Manual validation further reveals 26 identified incomplete patches that directly introduce new LFVulns, which we responsibly disclosed, resulting in 19 CVEs and $28,100 in bug bounties (§6.3).

2 Background & Motivating Example

This section delves into the background on Link Following vulnerabilities (LFVulns) and their patching mechanisms. We further present a real-world case to illustrate the inherent difficulty of identifying such patches in practice and how an ostensibly correct fix can evolve into a new exploitable vulnerability.

2.1 LFVulns and Incomplete Patch

Link Following vulnerabilities (LFVulns) refer to a class of security flaws in which an attacker leverages symbolic links or other reparse point [13] mechanisms to redirect a privileged program's file operations to security-sensitive locations. When a privileged application performs file operations without correctly validating the final resolved path, its operations may be inadvertently redirected to attacker-controlled targets.

For example, consider a privileged program that deletes a temporary file such as `C:\Windows\Temp\a.tmp` without validating the path. Because attackers typically have write access to temporary directories, they can replace this file with a symbolic link pointing to a protected file under `C:\Windows\System32`. In this case, the privileged deletion request will follow the link and operate on the protected file, potentially causing a persistent denial-of-service [14]. Given that file operations are ubiquitous across modern software systems, LFVulns arise in a wide range of applications and remain a persistent and impactful class of security issues.

Over the past decade, thousands of LFVulns have been patched across commercial software ecosystems, yet a substantial fraction of these fixes have failed to correctly address the underlying root cause on their first attempt. Since 2019, more than 150 incomplete patches have been documented that, while intended to fix a specific issue, inadvertently introduced new bugs or weaknesses, many of

which were quickly bypassed. For example, the Windows Error Reporting service [15] alone exhibited over 20 LFVulns between 2019 and 2022 [16], illustrating the persistent fragility and instability of LFVuln patches in real-world systems.

However, understanding and locating LFVuln patches remains equally challenging in practice. Commercial closed-source applications typically integrate security fixes into routine version updates rather than shipping standalone patches, causing security-relevant changes to be buried among extensive functional modifications. Patch advisories rarely disclose the vulnerability root cause or remediation logic, resulting in opaque patch semantics, and binary-only distributions offer no source-level diffs to support precise localization. These difficulties are further exacerbated by the lack of unified security guidelines and the highly diverse implementations of file operations across programs, leading developers to adopt inconsistent and often incompatible repair strategies. Consequently, LFVuln patches exhibit substantial variability in quality, making incomplete fixes both common and difficult to reliably identify.

```
1  int sub_E32240() {
2  +++ void* this = sub_E368B0();
3      // Patch attempts to block reparse-point paths by
           checking FILE_REPARSE_POINT (0x400)
4  +++ bool v7 = (((DWORD (__thiscall*)(void*))( *(DWORD*)
       this + 8 ))(this) & 0x400) != 0;
5  +++ if (!v7) {
6          DeleteFile(file); // Delete User-controllable
               file
7  +++ }
8  }
9  void* sub_E368B0() {
10     // Initialize vtable for FileSystem operations
11     *this = &AdwCleaner::FileSystem::vftable;
12     return this;
13 }
14 // Indirect virtual call: (*(DWORD*)this + 8)
15 int sub_E368F0(void* this) {
16     return GetFileAttributesW(this->file);
17 }
```

Listing 1: Code snippet of the incomplete patch for CVE-2023-28892 that ultimately led to CVE-2025-6****

2.2 Motivating Example

As shown in Listing 1, we present a real-world incomplete patch extracted from Malwarebytes AdwCleaner [17]. In this case, the patch was shipped as part of a feature update release that modified more than 150 functions, making the security-relevant changes difficult to distinguish from ordinary functionality updates.

Patch semantics. The patch is located in `sub_E32240`, where the code first constructs a FileSystem object via `sub_E368B0` (line 2), which sets the object's vtable pointer. To determine whether the target path corresponds to a reparse point [13], the patch invokes a virtual method through the vtable (line 4). This dispatch resolves to `sub_E368F0` (line 15), which calls `GetFileAttributesW` [18] API to obtain the file's attribute flags. The returned value is then checked with 0x400 (i.e., `FILE_REPARSE_POINT` [19]). Only when this flag is not set does the privileged program proceed to delete files via `DeleteFile` [20] API (line 6). Owing to language-specific implementation complexity, the patch is not immediately self-evident. Nevertheless, the intended purpose is to prevent the deletion routine from following symbolic links to protected system files.

Patch deficiencies. At first glance, the patch appears to prevent the privileged program from performing unchecked deletions, thereby ensuring that deletion requests do not follow symbolic links. However, the protection is applied only during attribute inspection and does not cover the subsequent deletion. This incompleteness stems from the fact that the attribute check and the subsequent deletion are performed as two independent operations, each operating on its own path resolution rather than a shared file handle. As a result, the patch provides no guarantee of atomicity between validation and use: the filename validated during the check may no longer refer to the same underlying file by the time the deletion routine executes. An attacker can exploit the resulting TOCTOU [21] window to redirect the deletion target after validation: attackers first create a benign file and acquire an opportunistic lock [22] on it, which allows a user-defined callback to be triggered upon specific file operations. This enables the file to first pass the attribute check (line 5). When the privileged program later attempts the deletion (line 6), the lock callback is triggered, giving the attacker a time window to replace the original file with a symbolic link, thereby bypassing the mitigation and resulting in a new LFVuln.

3 Overview

This paper aims to systematically identify and analyze incomplete patches in LFVulns, construct a taxonomy of their characteristic patterns, understand how these flaws give rise to new security risks, and explore how they should be properly addressed in practice. To achieve these goals, we answer the following research questions:

RQ1. How can we efficiently identify LFVuln patches in closed-source applications? A prerequisite for analyzing incomplete LFVuln patches is the ability to reliably identify the corresponding patches. Prior work [16, 23, 24] has examined such patches only on a case-by-case basis, relying heavily on manual inspection and ad hoc reverse engineering, which fundamentally limits scalability and prevents large-scale analysis of real-world commercial software. In §4, we develop *LinkDiff*, a binary differential-analysis tool that identifies LFVuln patches by analyzing changes along file-operation call chains and using an LLM-driven agent to reconstruct file operation semantics of modified code, thereby leveraging the LLM's contextual reasoning ability to handle indirect calls and other complex semantics that fall within the inherent limitations of traditional static analysis. *LinkDiff* uses a bottom-up strategy to isolate code fragments related to file operations and then employs the agent to recover their execution semantics, including contextual behavior and indirect control-flow interactions that static heuristics cannot capture. By recovering the semantic meaning of code differences, *LinkDiff* enables identification of LFVuln patches.

RQ2. What is the taxonomy of incomplete LFVuln patches, and what criteria distinguish the different categories and their characteristic behaviors? In §5, we conduct a systematic empirical analysis of historical incomplete LFVuln patches and employ *LinkDiff* to assist in both identifying these patches and analyzing their semantics. Our goal is to uncover the underlying

root causes and to understand why they frequently introduce new vulnerabilities. In total, we analyze the details of 60 incomplete patches from closed-source applications collected over the past six years. According to the analysis, we classified them into three categories consisting of 9 distinct types, along with root-cause analysis

RQ3. How prevalent are incomplete LFVuln patches among the latest released patches, and how are they distributed across our taxonomy? In §6, we assess whether the incomplete patch types identified by our taxonomy still appear in contemporary software. To enable this evaluation, we leverage retrieval-augmented generation to incorporate taxonomy-derived patch types as an external knowledge base [12] into *LinkDiff*, allowing it to directly identify and classify incomplete LFVuln patches in recent software versions. Using this enhanced version of *LinkDiff*, we analyze 76 recently released LFVuln patches from 62 closed-source applications. Our results show that incomplete patches remain widespread: more than 82% (63/76) of the latest fixes fall into one of our taxonomy categories. We further manually validated that 26 of these identified incomplete patches can be bypassed to yield new LFVulns, and we responsibly reported them to the affected vendors, obtaining 19 CVEs and $28,100 bug bounty rewards.

RQ4. How do incomplete patches lead to new LFVulns in real-world products? In §7, we examine two representative cases that we reported to vendors. For each case, we analyze the root causes of the incomplete fix and demonstrate how the flawed patch logic can be bypassed to yield new exploitation vectors with significant security impact. In addition, in §8.3, we propose six development principles for correctly implementing LFVuln mitigations from multiple perspectives.

4 Design and Implementation

4.1 Workflow

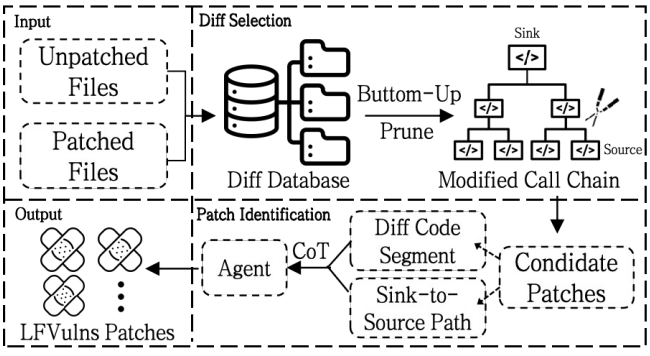

Figure 1: The workflow of *LinkDiff*

To efficiently and accurately identify LFVuln patches in closed-source applications, we design and implement *LinkDiff*, a differential-analysis tool tailored for identifying LFVuln patches. *LinkDiff* first narrows the analysis scope by isolating code differences that occur along file-operation call chains, thereby selecting patch candidates that are likely to influence file-system behaviors. It then employs an LLM-driven agent to perform context-aware semantic interpretation of these differences, enabling the recovery of their execution intent and the precise identification of LFVuln patches. The overall workflow is illustrated in Figure 1.

4.2 Diff Selection

In binary analysis, particularly for large closed-source commercial applications, performing whole-program differential analysis over all modified code has been shown to be highly inefficient [25][26][27]. Such global diffing incurs substantial analysis overhead and produces large amounts of semantically irrelevant changes, which obscure the modifications related to LFVuln patches and ultimately make the analysis unsustainable in practice. To address this challenge, the key insight behind *LinkDiff* is that LFVuln patches tend to appear along file-operation call chains. Based on this insight, *LinkDiff* performs semantics-guided differential analysis along file-operation call chains.

In this stage, *LinkDiff* first constructs a differential database by using BinDiff [28] from the pre-patch and post-patch binaries, which records modified, added, and removed functions along with their associated control-flow graphs and instruction sequences. This database serves as the foundation for identifying code regions that have potentially changed as part of the LFVulns fix. Next, to focus the analysis, *LinkDiff* constructs file-operation call chains using a bottom-up approach. Specifically, it first predefines a set of file-system APIs as sinks (e.g., `DeleteFile` [20]). These sinks are not program-defined APIs that depend on debug symbols, but imported Windows system APIs (e.g., from `kernel32.dll`). Therefore, even if a binary is stripped or name-obfuscated, file operations must still invoke these imported APIs. *LinkDiff* then performs a breadth-first search to traverse the caller hierarchy starting from these sinks using IDA [29]'s cross-references (i.e., *Xref* API). Here, IDA APIs serve only as an implementation backend for constructing file-operation call chains; similar functionality can also be supported by other decompilation frameworks such as Ghidra [30]. The search continues recursively, enumerating all functions that call the predefined sinks, and proceeds upwards in the call chain until no further callers are found. This enables *LinkDiff* to fully enumerate all call chains leading to these sinks. To further reduce the analysis scope, *LinkDiff* applies pruning by discarding any call chain that contains no modified functions. For the remaining chains, *LinkDiff* extracts the differential entries from the database and treats them as candidate patch locations. This targeted strategy greatly reduces analysis complexity while still maintaining broad coverage of potential patches.

A key detail is that *LinkDiff* recursively inspects all callees of each function on the call chain to detect modifications that may not appear directly on the chain itself. This is necessary because developers may encapsulate the actual fix within a helper function invoked at the same level upstream of the sink, while leaving the overall call chain structurally unchanged. To prevent unbounded recursion, we limit the maximum recursion depth to three, which empirically avoids combinatorial explosion while preserving the ability to capture such refactored or non-obvious fixes.

4.3 Patch Identification

The lack of source-level information and the use of object-oriented designs [31] in modern binaries lead to widespread function-pointer

usage and virtual dispatch, producing indirect control flow that obscures semantics and complicates static analysis [27, 32, 33]. Let's revisit Listing 1, where the patch issues a vtable-based indirect call resolved through a function pointer at a specific offset. Recovering such dispatches and their calling contexts is labor-intensive, requiring inference of object layouts, vtable structures, and compiler-specific conventions. Recent advances show that LLMs significantly enhance semantic reasoning in program analysis [34][35][36]. However, fully capturing file-operation semantics in binary analysis requires supplying the LLM with extensive code slices that cover all relevant control-flow and data-flow contexts. This process is both time-consuming and cumbersome, especially when indirect calls and complex control flows are involved. Thus, the key insight of *LinkDiff* is to tightly integrate LLM-based reasoning with binary analysis via function calling, enabling the LLM to reason over concrete execution contexts extracted from analysis results.

At this stage, *LinkDiff* leverages one-shot learning [37] in combination with Chain-of-Thought (CoT) reasoning [38] to guide the LLM in analyzing patch candidates. By utilizing function calling to enhance its understanding of execution semantics, the LLM can focus on interpreting the differential code fragments and the associated file-operation call chains that are relevant to LFVuln patch analysis. The agent in *LinkDiff* consists of three main components, and its prompt design (illustrated in Appendix C) encapsulates these components in a unified, task-oriented template:

- **One-shot learning** enables the LLM to perform similar reasoning tasks by mimicking the logic implemented in example-driven tasks, thereby understanding common repair patterns for LFVulns. In *LinkDiff*, these examples consist of publicly available LFVuln patch descriptions and their corresponding patch details, allowing the LLM to comprehend the common validation techniques employed by developers in file operations.
- **CoT** guides the LLM through a step-by-step reasoning process to refine the analysis of potential patches. This process includes: (1) locating the diff within the call chain and making an initial inference about its intent; (2) invoking provided functions to resolve the concrete targets of indirect calls (e.g., function pointers), thereby clarifying the patch semantics; and (3) reporting the patch when its observed semantics closely align with those of common LFVuln repair patterns.
- **Function Calling** enables more flexible contextual understanding by the LLM. By exposing IDA's code-analysis capabilities and differential-database queries as callable functions (as listed in Appendix D), *LinkDiff* avoids the need to supply large sliced code contexts to the LLM, a limitation common in traditional static LLM-based analysis, thereby making the analysis process significantly more efficient and adaptable.

5 Empirical Analysis and Taxonomy

To enable our empirical study, we first collect a set of historical incomplete LFVuln patches and then analyze them with *LinkDiff* to understand why such fixes remain susceptible to bypass (in §5.1). Based on our empirical analysis, we classify these incomplete patches into nine distinct types, which we further group into three overarching categories: *Under-Specified Path Validation*, *Flawed File-Metadata Assumptions*, and *Insufficient Privilege Isolation* (in §5.2). Each category is discussed in detail in the following subsections.

5.1 Incomplete LFVuln Patch Collection

To construct a dataset of historical incomplete patches for empirical analysis, we adopt a vulnerability-driven collection methodology. Note that we consider a patch to be incomplete when a program initially affected by an LFVuln receives a fix, yet subsequent versions continue to exhibit a vulnerability described by the same or a highly similar security advisory. We begin by identifying LFVulns from public vulnerability repositories. Using the CVE database, we perform an advanced search for entries containing the keywords *"Link Following"*, *"Local Privilege Escalation"*, and *"Symbolic Link"* over the period from 2018 to 2025. Each CVE is then cross-validated through the National Vulnerability Database (NVD) [39]. If its associated CWE falls within CWE-{59,63}, we include it; otherwise, we manually verify whether the vulnerability corresponds to LFVuln. Next, we extract CVE sequences corresponding to the LFVuln in the same product, where each CVE targets a different affected version and represents a subsequent bypass of earlier fixes. Such cases indicate that the earlier fix did not fully resolve the underlying LFVuln and consequently allowed a subsequent vulnerability to emerge. Through this two-step process, we collect a total of 108 target LFVulns, each reflecting the presence of an incomplete patch that resulted in follow-up LFVulns in later releases.

After determining the affected and fixed versions for each target LFVuln, we attempted to obtain and install both versions of the corresponding software packages. Using *LinkDiff*, we analyzed all modules (executables and libraries) that exhibited code differences to verify incomplete patches associated with each vulnerability. Due to practical constraints, such as commercial licensing, closed-source distribution, and the unavailability of older software versions, *LinkDiff* successfully analyzed 60 incomplete patches. It is worth noting that successive LFVulns with highly similar advisories do not necessarily imply that the earlier patch is incomplete, as they may occasionally correspond to two distinct LFVulns in the same product. To avoid this threat, we manually examined each candidate sequence to determine whether the later CVE indeed represents a bypass of the earlier fix rather than an unrelated vulnerability instance. In our dataset, all retained cases satisfy this criterion.

We argue that this dataset is highly representative for three reasons. First, the number of incomplete patches we analyze substantially exceeds all previously reported and publicly documented cases. Second, most of the corresponding LFVulns and their fixes lack publicly available technical details, indicating that our dataset covers largely unexplored instances rather than well-studied examples. Finally, we made a best-effort attempt to systematically collect and examine every potential incomplete patch released between 2018 and 2025. The entire process required approximately 20 person-days of effort from two authors.

5.2 Taxonomy

In this section, we present a detailed taxonomy of incomplete LFVuln patches, examining their underlying root causes and explaining why these fixes are incomplete and pose security risks.

Table 1: Taxonomy of Incomplete LFVuln Patches

Incomplete Patch Taxonomy	Root-Cause Description	CVEs
A. Under-Specified Path Validation		
A1. Localized Fixes	Patch addresses only a specific file-operation path, leaving equivalent operations unprotected.	CVE-2025-{27727,21373}, CVE-2024-{43551,30033,21432,12552}, CVE-2023-{32163,28892,21752,21542}, CVE-2022-21838, CVE-2021-{34484,26889,26866,26426}, CVE-2020-11474, CVE-2019-{1342,1339,1315,0863}
A2. Misaligned-Layer Fixes	Validation is applied at higher layers while low-level file primitives remain unchecked.	CVE-2024-49107
A3. Non-Atomic Check	Validation and file use are non-atomic, leaving exploitable TOCTOU windows.	CVE-2024-{9766,30093}, CVE-2023-36399, CVE-2022-{44704,43293,38604,26904,22718,21999,21919,21895}, CVE-2021-{43238,43237,26873}, CVE-2020-{1088,1048,0779,0754,0753}
A4. Missing Path Normalization	Paths across namespaces are not consistently normalized.	CVE-2024-28916, CVE-2020-1337
B. Flawed File-Metadata Assumptions		
B1. Missing Exception Handling	Missing error handling when strengthening permission checks.	CVE-2022-41120, CVE-2020-12335
B2. Improper Exception Handling	Error-handling paths introduce unsafe file operations.	CVE-2020-1337
B3. Insufficient Filename Randomization	Insufficient filename randomization leaves paths enumerable.	CVE-2025-25230, CVE-2019-1037
C. Insufficient Privilege Isolation		
C1. Localized Privilege Isolation	Privilege reduction is applied only to specific file operations rather than globally.	CVE-2025-21331, CVE-2024-26158, CVE-2023-{36394,36047,36046,35347,32053}, CVE-2021-{41379,26862}, CVE-2020-{17014,17001}
C2. Over-Privileged Isolation Contexts	Isolated contexts retain excessive privileges.	CVE-2023-{28868,28869}

Table 1 provides a detailed list of all the analyzed incomplete LFVuln patches and their classifications.

5.2.1 Under-Specified Path Validation. Patches in this category attempt to remediate LFVulns at the operation-primitive layer by validating file paths before performing sensitive operations or by correcting hazardous file-operation logic. However, their effectiveness is limited by incomplete path validation, leaving gaps through which adversarial path manipulation can evade the intended checks. As detailed below, this category comprises four specific subtypes.

Localized Fixes (A1). This class of patches applies a fix only to the specific file operation involved in the reported vulnerable location, without propagating the remediation to other semantically equivalent operations or to the broader execution context. Such localized fixes leave the protection inherently incomplete. Consistent with the "weakest-link" phenomenon [40], the system remains vulnerable as long as any alternative file operation with similar dangerous semantics is left unpatched. In addition, this subtype represents the largest proportion of all incomplete patches we analyzed, highlighting a broader lack of awareness regarding comprehensive LFVuln mitigation. In many cases, developers validate only the reported operation in the vulnerability, assuming that the program is fully protected thereafter, while overlooking other operations with similar hazardous semantics that remain exploitable.

To illustrate this issue, consider the patch for `CVE-2024-9244` [41], which is one of the incomplete patches analyzed by *LinkDiff* in the empirical study and ultimately resulted in the subsequent vulnerability `CVE-2024-48618` [**?**]. As shown in Listing 2, the patch introduces a reparse-point check only before the deletion operation, while the file write operation in the same execution context remains entirely unprotected. By redirecting exploitation to this uncovered pathway, attackers can circumvent the fix entirely.

```
// Patched File Deletion Operation
BOOL SafeDelete(LPCWSTR FileName) {
+++ if (!IsReparsePoint(FileName)) {
        DeleteFileW(FileName); // Protected deletion
+++ }
}
/* Unpatched File Operations in the Same Context */
...
while (TRUE) {
    BuildPath(PathBuffer, L"Foxit\\FoxitData.txt");
    // No reparse-point validation
    if (_waccess(PathBuffer, 0) == 0) {
        // Unprotected creation
        CreateFileW(PathBuffer, GENERIC_WRITE, ...);
        break;
    }
}
```

Listing 2: Illustration of a localized fix on CVE-2024-9244: only delete operation is patched, while write operations in the same execution context remain unprotected.

Misaligned-Layer Fixes (A2). The root cause of this subtype lies in a mismatch between the layer at which the patch is applied and the layer where the vulnerability actually originates. In practice, LFVulns arise from insufficient validation within a lower-level library function or a reusable file-operation primitive, yet the patch is applied only at an upper-layer caller or application-level logic, leaving the underlying primitive unprotected. Unlike localized fixes (A1), security risk in this subtype arises in different execution environments. Since the lower-level file-operation primitive is reused across multiple programs, a misaligned-layer fix allows an attacker to shift exploitation to another execution environment that invokes

the same unpatched primitive, thereby bypassing the mitigation applied only at the upper layer of the original vulnerable application.

In our analysis of the patch for CVE-2024-49107 [42], we observed that the fix was applied directly to the main executable WmsSelfHealingSvc.exe, where the fix relies on enabling process-level policy RedirectionGuard [43] (as discussed later). However, the vulnerable deletion operation is actually implemented in library WmsConfigTasks.dll. As a result, any other privileged program that imports and invokes the same library function remains exposed. This misaligned-layer fix ultimately led to a new LFVuln CVE-2025-54116 [44], in which an attacker can fully bypass the intended mitigation by exploiting an alternative execution environment, specifically the privileged program LogCollector.exe, which continues to call the unpatched library function.

Non-Atomic Check (A3). This subtype arises when the patch performs path validation and the subsequent file operation in a non-atomic manner. In these cases, the fix first validates the target path and then reopens the file to execute the operation, resulting in a temporal gap between the check and the actual action. This separation introduces TOCTOU, since the validated file may no longer correspond to the file performed upon during the operation. Such non-atomic designs allow attackers to alter the file state between the check and the use, thereby bypassing the intended mitigation. Notably, this subtype appears frequently in our analysis. One key reason is that the atomicity requirements of secure file operations are inherently non-obvious and often counterintuitive. In practice, modern operating systems expose file APIs whose security guarantees rely on precise handle reuse and invariant path resolution, whereas developers frequently adopt path-based checks that do not guarantee atomic semantics.

```
1  BOOL PatchedWrite(LPCWSTR path, BYTE* buf, DWORD len){
2  +++ if (!CheckWritableByLowPriv(path)) return FALSE;
3      HANDLE h = CreateFileW(path, GENERIC_WRITE, ...);
4      if (h == INVALID_HANDLE_VALUE) return FALSE;
5      BOOL ok = WriteFile(h, buf, len, NULL, NULL);
6      return TRUE;
7  }
8  /* Low-privilege writability check */
9  BOOL CheckWritableByLowPriv(LPCWSTR path){
10     // Switch to low-privilege context
11     ImpersonateClient();
12     HANDLE t = CreateFileW(path, WRITE_ATTRIBUTES,...);
13     if (t == INVALID_HANDLE_VALUE) return FALSE;
14     return TRUE;
15 }
```

Listing 3: Simplified patch logic of CVE-2020-1048.

Let us examine the patch for CVE-2020-1048 [45] in Listing 3. Although the fix appears intuitively sound—permitting the privileged write only when the file is not writable by a low-privilege context, it fails to satisfy the atomicity requirements of secure file operations. Both the pre-check (Line 2) and the subsequent privileged write (Line 5) rely on path-based file APIs rather than reusing the same file handle. Because Windows resolves the path anew for each CreateFileW [46] invocation, the object validated during the low-privilege impersonation phase may differ from the object later opened for writing. This non-atomic design introduces a TOCTOU: an attacker can replace the file with a symbolic link in between the check and the use, redirecting the privileged write to an unintended target. This case illustrates that path-based validation, despite its intuitive appeal, does not ensure atomic semantics and results in patches that remain vulnerable to race-condition bypasses.

Missing Path Normalization (A4). This subtype occurs when patches apply validation or policy checks to attacker-controlled paths without first normalizing them into a canonical namespace. In Windows, file path resolution is fundamentally based on the NT path namespace [47]. Consequently, most file-system security mechanisms implicitly assume by default that the input path belongs to the NT path. However, when attacker-controlled paths are accepted without normalization, the same file object may be expressed in alternative namespaces (e.g., UNC path [48]), causing the patch to be silently bypassed. To further illustrate this, we examine two representative forms of patch implementations.

1) RedirectionGuard [43] is a process-level mitigation that is enabled per process via *SetProcessMitigationPolicy* [49]. When activated, the application itself rejects any reparse-point redirection initiated by non-administrative users. However, when the path is fully attacker-controlled, and no normalization is applied, an adversary can supply alternative namespace representations such as \UNC\path. Without canonicalization, the program treats this input as equivalent to an NT path, but Windows resolves UNC paths through the system service, which does not have RedirectionGuard enabled. Consequently, the final resolution step is implicitly delegated to a process outside the mitigation boundary, causing the protection to become ineffective for UNC-style inputs even when the originating application has explicitly enabled this mitigation.

2) GetFinalPathNameByHandle [50] is a Windows API that returns the resolved path of a given file handle. In our analysis, developers frequently adopt this API as a straightforward mitigation. For example, in the patch for CVE-2020-1337 [51], we observe this issue (see Appendix E), the program opens a file using path F_a, obtains a handle, and then invokes the API to retrieve the resolved path F_b. The patch subsequently checks whether F_a and F_b refer to the same location. Under the NT path namespace, this approach is sound, as it preserves atomicity and reliably detects redirections. However, when the input is supplied in UNC path, the API does not resolve the handle to an NT path and instead returns the original UNC string unchanged. As a result, $F_a = F_b$ holds for all UNC inputs, even when redirection has occurred, allowing the mitigation to be bypassed and ultimately leading to CVE-2020-17001 [52].

5.2.2 Flawed File-Metadata Assumptions. Patches in this category attempt to remediate LFVulns at the access-control layer by enforcing strict permissions on security-sensitive files or directories, thereby preventing attackers from creating or substituting symbolic links. However, flaws in how these access-control operations are applied or managed can undermine the intended protection. As detailed below, this category consists of three subtypes that capture the primary patterns through which such fixes remain incomplete.

Missing Exception Handling (B1). In this subtype, developers aim to prevent attackers from creating or substituting symbolic links by enforcing strict access-control checks on files involved in dangerous file operations. However, they omit dedicated error-handling or rollback logic when applying these policies, leaving

the patch dependent on unchecked assumptions about the file's metadata state. Such patches commonly assume that the target file does not yet exist and fail to consider adversarial conditions, such as the file already being present or being exclusively held by another process. Consequently, when an attacker precreates or monopolizes the expected path, the access-control operation raises an exception that is silently ignored, causing the intended protection mechanism to be skipped and ultimately rendered ineffective.

To illustrate this issue, consider the patch of `CVE-2024-43114` [53] in Appendix F. The patch creates the target directory during the program installation phase and subsequently performs an access-control enforcement step that prevents any unauthorized symbolic link creation to the directory. Clearly, this patch is incomplete because the access-control enforcement occurs after the directory creation, without any error handling for potential failures in the creation process. As a result, any scenario that causes the directory creation to fail can bypass the access-control step. For example, if an attacker pre-creates the directory in advance, the creation attempt fails, rendering the intended protection ineffective.

Improper Exception Handling (B2). This subtype arises when the patch introduces improper exception handling in the access-control enforcement. Specifically, while attempting to enforce access control on file operations, the patch may improperly handle exceptions, such as in cases where it attempts to clean up temporary files upon failure. In such cases, this cleanup operation may bypass necessary access-control checks and inadvertently introduce vulnerable file operations. As a result, the patch itself can become a new attack surface, undermining the intended protection and allowing attackers to exploit flaws in the exception-handling process.

As shown in Listing 4, the patch introduces an access-control check prior to the `vulnerable_file_operations` function, along with an associated exception-handling path intended to abort the operation and remove related files upon check failure. However, because the cleanup directly invokes `DeleteFile` without performing any additional security checks, it introduces a new unsafe file operation, causing the patch itself to become a new attack vector. As a result, the patch inadvertently introduces a new LFVuln (CVE-2020-17014 [54]), which can be exploited by attackers who create a same-named symbolic link to deliberately cause the access-control check to fail and force the program to enter the cleanup.

```
1 STATUS v18 = CheckPathLinkAccessPolicy(filePath);
2 if (v18 < 0) { /* Adding exception handling */
3     DeleteFileW(filePath); /* exception-path cleanup */
4     goto LABEL_36;
5 }
6 LABEL_36:
7     return FALSE;
8 /* Vulnerable file operations follow */
9 vulnerable_file_operations();
```

Listing 4: Code snippet of the patch in CVE-2020-1337.

Insufficient Filename Randomization (B3). Developers often attempt to mitigate LFVulns by randomizing file names, such as using timestamps, in order to reduce an attacker's ability to create symbolic links. Such filename randomization is intended to indirectly strengthen access control by making target paths harder to predict. However, when the randomization scheme is insufficient, the generated names may remain predictable or enumerable in practice. For instance, in an LFVuln patch for the *Windows Error Reporting* service (`WerSvc`) [15], we observed that developers relied on `GetTempFileName` [55] API to generate a temporary filename. However, this API generates filenames with a fixed prefix and a four-digit hexadecimal suffix (e.g., `WER_78af`), yielding a filename space of size $|\mathcal{S}| = 16^4$, i.e., only 16 bits of entropy, which substantially limits the resistance to brute force. In practice, an attacker can enumerate this space and create symbolic links for all possible candidates, thereby reliably bypassing the mitigation. The subsequent fix no longer relies on randomization alone, but instead enables the process-level mitigation *RedirectionGuard* for the `WerSvc` service. The root cause of this incompleteness is that such randomization only reduces the probability of a successful attack, rather than eliminating it altogether.

5.2.3 Insufficient Privilege Isolation. Patches in this category attempt to mitigate LFVulns by restricting the privileges of file-operation principals via execution-environment isolation. However, insufficient or misapplied isolation allows file operations to be invoked from unprotected contexts or executed with residual privileges, undermining the intended security guarantees. As discussed below, this category comprises two distinct subtypes that capture the primary ways in which privilege isolation remains incomplete.

Localized Privilege Isolation (C1). This subtype arises when a patch introduces privilege-isolation mechanisms, such as impersonation [56] or privilege dropping [57], to restrict file operations, but applies the isolation only to a limited subset of file primitives or code paths. Consequently, attackers can bypass the intended mitigation by redirecting exploitation toward file operations that remain unisolated, achieving the same dangerous effects without interacting with the protected paths. The root cause is the failure to enforce privilege isolation at the level of the global execution environment: partial isolation inevitably leaves alternative file-operation pathways outside the isolation boundary exposed to abuse.

```
1  /* Read file as SYSTEM */
2  buffer = ReadUserFile_AsSystem(
3      L"C:\\Users\\<username>\\AppData\\Local\\...\\source.jpg");
4  /* Patch: perform write under logged-on user context */
5  if (ImpersonateLoggedUser(token)) {
6      StoreBuffer_AsImpersonatedUser(
7          L"C:\\Users\\Public\\Captures\\...\\target.jpg", buffer);
8      RevertToSelf();
9  } else {
10     Log("impersonation failed, abort write");
11 }
```

Listing 5: Code snippet of the patch in CVE-2023-36047.

Listing 5 illustrates the patch for `CVE-2023-36047` [58] which ultimately led to `CVE-2024-21447` [59]. The program impersonates a low-privilege user when writing to `target.jpg` (Line 5), preventing attackers from redirecting the write operation through a symbolic link placed at `target.jpg`. However, the patch applies privilege isolation only to the write operation and fails to isolate the preceding file read (Line 2). As a result, the `source.jpg` remains accessible under the original privileged context and can be replaced with a symbolic link to a sensitive file. This enables a confused deputy scenario in which the privileged program reads attacker-chosen sensitive data and subsequently writes it to `target.jpg` under reduced privileges, leading to an information leak.

Over-Privileged Isolation Contexts (C2). This subtype captures cases where a patch introduces privilege-isolation mechanisms that are applied consistently across relevant file operations, yet the resulting execution context retains excessive privileges and an incorrect integrity level. Although isolation is explicitly enforced, the isolated context still possesses sufficient authority to perform security-sensitive file operations. Under this weakened isolation, attackers can still exploit these primitives within the purportedly isolated context to trigger LFVulns or equivalent dangerous effects. For example, our empirical analysis reveals cases where patches reduce the execution context from SYSTEM to NETWORK SERVICE. Although this change appears to lower privileges, both accounts reside within the same trust boundary and retain access to sensitive system resources. Consequently, the isolation is largely superficial and does not eliminate the underlying security risk.

6 Evaluation

6.1 Experiment Setup

Latest LFVuln Patch Dataset. To evaluate *LinkDiff* and assess the prevalence of incomplete patches in contemporary LFVuln mitigations, we collected 76 latest LFVuln patches from 62 closed-source applications across 25 major vendors, including industry-leading manufacturers such as Microsoft, Samsung, Apple, and Intel. By focusing on the latest patches rather than historical cases, our evaluation examines whether incomplete LFVuln fixes remain common in today's real-world software. The inclusion of top-tier vendors further strengthens the significance of this analysis: if even these well-resourced vendors frequently deploy incomplete patches, it suggests that LFVuln patching remains a systemic challenge posing serious security risks to a large user base. Specifically, we follow a collection process similar to that described in §5.1, but focus on identifying and analyzing the *latest* LFVuln patches. We consider a patch *latest* if no subsequent LFVuln affecting the patched version has been reported, indicating that the fix has not been bypassed in later releases and remains deployed throughout the software lifecycle. For non-Microsoft applications, we manually obtained both the latest patched versions and their corresponding pre-patch versions from official vendor releases or trusted distribution channels, and evaluated them pairwise. For Microsoft Windows components, however, patches are not released as standalone updates but are bundled into monthly cumulative update packages. Accordingly, we adopt a three-step procedure to extract the relevant binaries. ❶ Given a specific update identifier (e.g., `KB5053594` [60]) related to a LFVuln, we download the official `.MSU` update package [61] released by Microsoft. ❷ We iteratively unpack the `.MSU` package to extract all updated executable files included in the release. ❸ Based on vulnerability advisories, we collect the executables corresponding to the LFVuln under analysis.

LinkDiff Enhancement. The original prototype of *LinkDiff* was designed to locate and analyze LFVuln patches, serving as a differential analysis framework for patch discovery and inspection. Our empirical study further reveals that incomplete LFVuln patches exhibit recurring structural patterns and root causes that can be systematically characterized. A key objective of this work is to empower developers without file-system security expertise to independently verify the correctness of LFVuln patches. To this end, we enhance *LinkDiff* by incorporating these empirical observations, allowing it not only to detect incomplete LFVuln patches but also to automatically classify them according to their underlying causes.

Concretely, we encode the taxonomy of incomplete LFVuln patches into *LinkDiff* as a domain-specific knowledge base. Each subtype is represented by a concise semantic description, supplemented with one or two representative real-world CVE cases and corresponding patch snippets that exemplify the underlying flaw. For each patch category, we further summarize the representative example code, the underlying cause of incompleteness, and the corresponding bypass mechanism in Markdown format. This enables the analysis agent to retrieve relevant patch patterns and contextual evidence during reasoning, thereby grounding LLM-driven decisions in concrete domain-specific information. This knowledge base is integrated into the agent-driven analysis pipeline and exposed to the LLM through structured prompting, allowing the model to reason about patch semantics with explicit guidance grounded in prior empirical evidence. By augmenting *LinkDiff* with this knowledge-driven component, the tool is elevated from a general framework for LFVuln patch identification and analysis to a semantics-aware analysis system capable of automatically identifying incomplete LFVuln patches and assigning them to specific subtypes. We think this enhancement is also particularly critical for large-scale analysis of newly released patches, as it enables systematic evaluation with minimal reliance on expert knowledge and manual effort.

Running Environment. We conducted experiments on a Windows platform equipped with a 2.1 GHz 28-core CPU and 32 GB RAM. For model selection, we utilized OpenAI's GPT-4o as the underlying large language model (LLM) within the agent of *LinkDiff*.

6.2 Evaluation Against Ground Truth

To assess the precision and recall of the enhanced *LinkDiff* in classifying incomplete LFVuln patches, we use the historical incomplete patches collected in §5.1 as ground truth. Since the true subtype labels of these patches have been established through our empirical analysis, this dataset provides a reliable benchmark for evaluation.

Result. The results are summarized in Table 2. *LinkDiff* correctly assigns the exact subtype to 49 of the 60 historical incomplete patches. Among the remaining cases, three are misclassified only at the subtype level within the correct top-level category, whereas eight are misclassified across top-level categories. Under our evaluation protocol, these results correspond to $P_{\text{tax}} = 94.23\%$ and $R_{\text{tax}} = 85.96\%$. We note that using real cases from the ground-truth set as RAG examples may introduce potential evaluation bias. To mitigate this threat, we deliberately restrict the knowledge base to only 1-2 representative examples per category, rather than all instances in the dataset, so that it provides limited category-level guidance instead of memorizing the full ground truth. In addition, *LinkDiff* also performs strongly on the latest-patch dataset in §6.3, which provides further evidence that its effectiveness is not merely due to overfitting to the historical dataset.

We subsequently manually examined the errors observed during evaluation and systematically analyzed their underlying causes. For diagnostic purposes, we adopt taxonomy-aware definitions of false positives and false negatives, rather than their conventional binary-classification meanings. Specifically, a misclassification within the correct top-level category is denoted as an intra-category false

Table 2: Taxonomy-aware classification performance of augmented LinkDiff.

Incomplete Patch Category	TP	FP_{intra}	FN_{inter}	P_{tax}	R_{tax}
Under-Specified Path Validation	35	2	5	94.59%	87.50%
Flawed File-Metadata Assumptions	2	1	2	66.67%	50.00%
Insufficient Privilege Isolation	12	0	1	100.00%	92.31%
Overall	49	3	8	94.23%	85.96%

positive (FP_{intra}), as it reflects an imprecise subtype assignment rather than a fundamental misunderstanding of the patch deficiency. A misclassification across top-level categories is denoted as an inter-category false negative (FN_{inter}), as it indicates a failure to identify the patch's underlying class of repair deficiency. Accordingly, we compute taxonomy-aware precision and recall as $P_{tax} = TP/(TP + FP_{intra})$ and $R_{tax} = TP/(TP + FN_{inter})$, respectively. For false positives, the most common scenario arises when a software has undergone multiple LFVuln fixes over time. Even though only the code differences associated with the current fix are provided to *LinkDiff*, the LLM may still incorporate patterns from earlier repairs when reasoning over file-operation call chains. This interference from historical repair patterns can distort the semantic interpretation of the current patch, leading *LinkDiff* to incorrectly classify the fix and produce false positives. On the other hand, false negatives primarily stem from the inherent hallucination tendencies of LLMs during complex program semantic reasoning. In scenarios involving deep call chains and implicit patch semantics, the model may misinterpret or overlook critical information, leading to deviations from the true incompleteness of a patch. Such errors reflect the current limitations of LLMs in fine-grained program understanding rather than systemic flaws in the design of *LinkDiff*.

6.3 Evaluation of Latest LFVuln Patches

As outlined in §6.1, we aim to investigate whether incomplete LFVuln patches remain prevalent in contemporary real-world software. To this end, we apply the enhanced *LinkDiff* to perform analysis over a dataset of the latest released LFVuln patches.

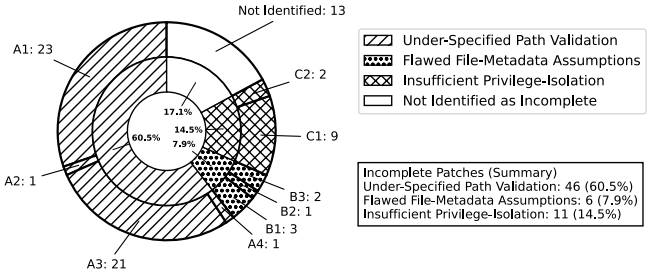

Figure 2: Hierarchical Distribution of the latest released LFVuln Patches

Result. Figure 2 presents the distribution of incomplete fixes identified in recently released LFVuln patches, together with their categorization across different root-cause classes. Overall, *LinkDiff* flags 63 patches as potentially incomplete. Manual validation confirms 56 of the 63 flagged cases and identifies the remaining seven as false positives. Among these incomplete fixes, *Under-Specified Path Validation* is the dominant category, with 46 cases (60.5%), suggesting that insufficient or partial path validation remains the primary source of patch incompleteness. In comparison, *Flawed File-Metadata Assumptions* account for 6 cases (7.9%), while *Insufficient Privilege-Isolation* contributes 11 cases (14.5%), reflecting a non-negligible presence of isolation-related issues. Only 13 patches are *not identified as incomplete*, and we further manually inspected each of these patches and confirmed that their fixes do not expose exploitable behaviors that could be leveraged by subsequent attacks, providing additional confidence in the soundness of *LinkDiff*.

Notably, the overall distribution of incomplete patches in these newly released patches closely resembles that observed in historical LFVuln patches, as revealed by our empirical analysis. This consistency suggests that the underlying causes of incomplete fixes have remained largely unchanged over time, and that developers continue to encounter similar pitfalls when mitigating LFVulns. Such persistence highlights the importance of systematic classification and automated identification of incomplete patches for LFVulns, enabling consistent reasoning about patch quality beyond ad-hoc or case-specific patching strategies.

Table 3: Real-World Impact of Incomplete Patches found by *LinkDiff* in Latest Patch Dataset

#	Latest Patch Cases	Vendor	Classification	Status
1	CVE-2020-9682	Adobe	A1	CVE-2025-6*****
2	CVE-2022-38699	ASUS	A1	CVE-2025-9***
3	CVE-2023-28892	Malware Bytes	A1	CVE-2025-6****
4	CVE-2023-39230	Intel	A1	CVE-2025-2****
5	CVE-2023-34997	Intel	A1	Confirmed
6	CVE-2024-6147	HP	A1	CVE-2024-6***
7	CVE-2024-7987	Rockwell	A1	CVE-2025-3***
8	CVE-2024-8424	WatchGuard	A1	CVE-2025-0***
9	CVE-2025-48443	Trend Micro	A1	CVE-2025-5****
10	CVE-2024-30377	G DATA	A1	CVE-2025-2***
11	CVE-2025-20099	Intel	A1	CVE-2025-2****
12	CVE-2025-27246	Intel	A1	CVE-2025-3****
13	CVE-2024-49107	Microsoft	A2	CVE-2025-5****
14	CVE-2025-32721	Microsoft	A2	Confirmed
15	CVE-2024-8404	PaperCut	A3	CVE-2024-8***
16	CVE-2024-30377	G DATA	A3	CVE-2024-5****
17	CVE-2024-26216	Microsoft	A4	Confirmed
18	CVE-2024-43114	JetBrains	B1	CVE-2025-5****
19	CVE-2024-23908	Intel	B1	Confirmed
20	CVE-2020-12335	Intel	B1	Confirmed
21	CVE-2023-35342	Microsoft	B2	CVE-2024-3****
22	CVE-2024-49051	Microsoft	B3	CVE-2025-2****
23	CVE-2024-6974	Cato Network	B3	Confirmed
24	CVE-2024-35254	Azure	B3	CVE-2024-3****
25	CVE-2024-45315	SonicWall	C2	CVE-2025-3****
26	CVE-2024-21397	Azure	C2	Confirmed

Real-World Security Impact. Beyond measuring the prevalence of incompleteness in the latest released LFVuln patches, we further assess their concrete security impact in practice. Guided by *LinkDiff* 's analysis outputs, we manually examined every reported case. Among them, we confirmed 7 false positives, which were mistakenly categorized as category-A, B, and C incomplete patches (3, 2, and 2 cases, respectively), mainly because *LinkDiff* conservatively

assumes that the target file of the fixing operation is fully attacker-controllable (e.g., the full file path). For the remaining cases, we found that 26 incomplete patches can be directly bypassed, resulting in new LFVulns. The others still constitute genuine incomplete patches; although they do not directly yield an exploitable bypass at present, they reflect clear repair deficiencies and non-negligible future exploitation risk, as further discussed in §8.1. An overview is provided in Table 3, the CVE listed in the first row denotes the vulnerability addressed by the latest patch, while the CVE reported in the disclosure status corresponds to the newly introduced LFVulns we reported. We responsibly disclosed these incomplete patches and their corresponding bypass techniques to the affected vendors, including major companies such as Microsoft, Intel, and Adobe. All reported issues were officially acknowledged and confirmed by the vendors, with 19 of them assigned CVE identifiers. In addition, our disclosures resulted in bug bounty awards totaling $28,100, underscoring that incomplete LFVuln patches constitute tangible and non-negligible security risks in real-world software systems.

6.4 Ablation Study

To evaluate our design choices, especially the impact of diff selection on *LinkDiff* 's effectiveness, we conduct an ablation study by comparing the following configurations:

- **Full Setup (Baseline):** The complete *LinkDiff* described in §4
- **Global Diff Analysis:** Instead of analyzing only file-operation call chains selected by the diff selection process, this variant considers all code differences, assessing the role of diff selection in reducing analysis overhead.

We record the analysis results of each setup on 60 historical incomplete patches (i.e., ground truth), and summarize the statistics in Table 4. For each setup, we measure the key metrics required to fully analyze an incomplete patch, including the time for diff extraction, the number of candidate diff functions to be analyzed, the end-to-end analysis time of *LinkDiff*, as well as the token consumption and monetary cost incurred by the LLM.

As shown in Table 4, removing diff selection substantially increases the analysis burden. Compared with the baseline, the diff extraction time increases by 43%, and the number of candidate diff functions rises by 805%. This sharp growth in candidate code significantly enlarges the analysis context passed to the LLM-based agent, making the downstream reasoning stage much more expensive.

Table 4: Ablation study of *LinkDiff* against the ground truth.

Setup	Diff Extract Time (s)	Diff Functions	Analysis Overhead (Time / Tokens / Cost)
Full Setup	19.42	36.49	3.4min / 725*k* / $0.29
Global Diff	27.77	330.32	*Timeout*
Avg. Increase	↑ 43.0%	↑ 805.2%	–

Consequently, the *Global Diff* setup fails to complete the analysis for all incomplete patches. The excessively large code context causes the agent to exceed its context and time budget, so the resulting analysis is inherently partial and incomplete. We therefore omit the corresponding time, token, and cost metrics in Table 4, as they are no longer meaningful or comparable. This result demonstrates that diff selection is not merely an overhead-reduction optimization, but a critical design choice for enabling *LinkDiff* to perform effective and complete analysis on real-world cases.

7 Case Study

In this section, we analyze two vendor-reported incomplete LFVuln patches and their bypass exploits, showing that incomplete mitigations can lead to new LFVulns with significant security impact.

Case A: Windows Image * Service.** *LinkDiff* identifies the latest LFVuln patch that addresses CVE-2023-35342, as shown in Figure 3, and classifies it as *Improper Exception Handling* (B2). As highlighted in the green box, the patch attempts to detect symbolic links by checking the input `FileName` (Line 4). When a symbolic link is encountered, the program treats the path as invalid and enters an error-handling branch (Line 6). However, during subsequent cleanup, the file is deleted without any further validation (Line 7), introducing a security flaw in the exception-handling path. This allows an attacker to trigger the error-handling path by creating a symbolic link named `w**.log`. By exploiting the oplock to replace the file before the privileged cleanup operation, the deletion can be redirected to an attacker-controlled target, resulting in arbitrary file deletion and local privilege escalation. We promptly reported this issue to Microsoft, which subsequently acknowledged and assigned CVE-2024-3****, awarding us a $2,000 bug bounty.

```
1  void sub_1800100AD() {
2    GetFilePath(L"C:\\Debug\\w**.log", &FileName);
3  + if (!CheckPathIsLink(FileName)){
4      ··· // unchanged logic
5  + }else{
6  +    DeleteFile(FileName);
7    }
```

Figure 3: Pseudo code snippet of patch for CVE-2023-35342

Case B: Adobe C* software.** In analysis of the latest LFVuln patch for CVE-2020-9682, *LinkDiff* finds that the patch enforces strict validation on only one file write operation, while overlooking another directory creation operation within the same execution context. As a result, the patch is classified as a *Localized Fix* (A1). This incomplete mitigation allows an attacker to shift the exploitation primitive from the protected write operation to the unchecked directory creation. Specifically, an attacker can pre-create a symbolic link for the target directory that points to an arbitrary path, and subsequently trigger the directory creation logic to follow the link, leading to arbitrary file creation and ultimately causing a denial-of-service security issue. We responsibly disclosed this incomplete patch and the corresponding exploit details to Adobe, which promptly acknowledged the issue, assigned CVE-2025-6****.

8 Discussion

8.1 Potential Security Implications

It is important to clarify the potential security implications posed by incomplete LFVuln patches. Such patches do not necessarily fail immediately or directly introduce new LFVulns. In other words, incomplete patches do not guarantee the emergence of vulnerabilities right away, but rather increase the likelihood of future exploitation.

For example, some programs attempt to fix LFVulns by addressing a single file operation. Due to developers' insufficient awareness of security measures for such issues, they may assume that simply eliminating the vulnerable operation ensures the current safety of the program. Factually, our observations show that such cases are not uncommon. These incomplete patches, which represent quick fixes, fail to resolve the underlying issue. Consequently, as the program evolves and new features are added, it may inadvertently introduce dangerous file operations. Developers may mistakenly believe the original patch has completely solved the problem and thus overlook potential future risks. As a result, such incomplete patches act like latent bombs, potentially causing new LFVulns to emerge with future updates to the program.

8.2 Responsible Use of LLM

Role of LLMs. We are fully aware of the well-known hallucination and instability issues of LLMs. Therefore, this work deliberately avoids using large language models as unconstrained analyzers for incomplete LFVuln patches. Instead, we adopt a disciplined agent-based design that leverages LLMs as controlled reasoning components rather than free-form generators. Specifically, *LinkDiff* exploits the agent's function-call capability to strictly constrain the model's actions to a predefined set of binary-analysis primitives, such as resolving function cross-references and inspecting virtual-table or function-pointer offsets in the compiled code. Rather than reasoning over decompiled code in isolation, the LLM reconstructs patch semantics by leveraging function calls to obtain sufficient execution context and structured analysis results.

The determinism of LLMs. Our goal is not to force LLMs into full determinism, but to use them in a controlled setting where limited output variability has minimal impact on the overall system. In *LinkDiff*, the LLM is confined to a relatively narrow role: it interprets structured analysis outputs and reasons over bounded patch contexts, rather than serving as an unconstrained analyzer for raw binaries. The core analysis remains grounded in binary-analysis function calls, which substantially reduces the room for unstable free-form generation. As a result, the system's behavior depends primarily on the agent's analysis capabilities, with the LLM acting as a constrained reasoning component. In addition, *LinkDiff* already performs strongly with GPT-4o, suggesting that the current results should be viewed as a conservative lower bound rather than one tied to the stronger available model (e.g., GPT-5.4).

8.3 Principles of Secure Patch Development

In this section, we propose six patch development principles, each with distinct focuses and trade-offs. Developers can select the most appropriate patching strategy based on the specific context and requirements of the vulnerability being addressed.

Secure Encapsulation of File Operations. This principle focuses on strengthening file operations encapsulated in common libraries by adding inline checks, ensuring that all callers inherit the same validation. By centralizing the validation logic within the library, we reduce the risk of inconsistent checks and ensure that all usages follow a uniform security policy.

Atomicity of Validation and Operation. This principle emphasizes that path validation and the corresponding file operation must occur atomically, meaning they should be performed on the same file handle within the same time window. This prevents attackers from exploiting potential timing windows between the validation and operation, ensuring the integrity of the check and the operation.

Standardized Path. The goal of this principle is to unify the handling of file paths under a single, standardized semantic namespace, thereby eliminating risks associated with parsing inconsistencies. By ensuring consistent resolution of all paths within the same namespace, we mitigate vulnerabilities arising from discrepancies in path resolution that underlie many LFVulns.

Explicit Authorization. When creating security-sensitive files, this principle emphasizes the importance of explicitly setting access control permissions instead of relying on inherited ones. By directly defining the permissions, we ensure that security controls are properly enforced and not inadvertently bypassed, addressing potential flaws in access control mechanisms (e.g., B1 and B2).

Strong Randomization and Unpredictability. This principle advises the use of strong, non-enumerable, random filenames for temporary or placeholder files. By using unpredictable names, the risk of attackers guessing or brute-forcing file names to create symbolic links is minimized, thus reducing the likelihood of LFVulns.

Least Privilege. For every file operation, the process should be granted only the minimal permissions necessary to complete that specific operation. This minimizes the attack surface by ensuring that even if a vulnerability exists, its potential impact is constrained by limiting the privileges granted to each operation.

8.4 Limitations

Objective of prototype. While not designed as a vulnerability discovery tool, *LinkDiff* serves as a validation tool to help developers verify the correctness of patches before deployment, preventing the introduction of secondary exploitations. However, despite its ability to identify and analyze issues in patches, its results must be interpreted in conjunction with the expertise of file system security professionals. In other words, *LinkDiff*'s output should serve as a complementary resource for developers' decision-making, rather than a substitute for human judgment.

Scope of prototype. We found that the majority of closed-source software with LFVulns is developed in C/C++. Therefore, *LinkDiff* was initially designed to support patch analysis for C/C++ software, and has not yet been extended to other programming languages. However, we believe that the methodology employed by *LinkDiff* can be adapted to any language that has decompilers and static analysis support. Additionally, *LinkDiff* faces limitations when analyzing software that has undergone code obfuscation (e.g., using OLLVM [62]), as LLMs typically perform poorly on such obfuscated code, and deobfuscation falls outside the scope of this work.

Extension to Open-Source Ecosystems. Our current study focuses on closed-source binaries, which are harder to analyze due to the lack of source code and limited semantic information. We believe that, if incomplete LFVuln patches can be localized and analyzed in this setting, the same goal should be more readily achievable in open-source ecosystems with public source code and explicit commit histories. Extending *LinkDiff* to open-source systems (e.g., Linux) would require shifting the analysis focus from binary-level

patch understanding to source-level logic analysis, which we view as a meaningful direction for future work.

Dataset Size. Our study analyzes 60 historical and 76 latest LFVuln patches. Although the datasets may appear limited in size, we made best efforts to systematically collect and analyze LFVuln patches in the wild, and the resulting patch dataset and analyzed details exceed the entirety of publicly documented LFVuln fixes. The dataset size is primarily constrained by the limited availability of historical binaries for closed-source commercial software and widespread code obfuscation. Nevertheless, the observed category patterns are consistent across vendors and patch generations, suggesting that our findings remain representative despite this limitation.

9 Related Work

Symbolic link security. Prior work has studied symbolic-link vulnerabilities across Windows, Android, and Linux [1–4, 63, 64]. In Windows, Kim et al. [63] analyze file-system semantic discrepancies, while JERRY [3], LinkZard [1], and FAVDisco [4] detect or exploit unsafe symbolic-link operations in privileged programs. In contrast, our work investigates whether the corresponding repair logic is complete and how flawed fixes leave or reintroduce security risks. Beyond Windows, PathSentinel [2] detects file-system vulnerabilities in Android, whereas SafeOpen [7], PolyScope [65], and JIGSAW [8] mitigate symbolic-link abuse through secure path resolution or access-control analysis.

While extensive research has examined symbolic-link vulnerabilities and corresponding mitigation techniques, existing work has largely overlooked the correctness of real-world patch implementations. In particular, there is still a lack of systematic understanding of how LFVuln patches fail in practice. Our work fills this gap by providing, to the best of our knowledge, the first systematic study of incomplete LFVuln patches, including a taxonomy of recurring incomplete-fix patterns, their root causes, and how they can still lead to bypasses. We view this systematic characterization as the primary contribution of the paper, as it sheds light on an important but underexplored aspect of file-system security.

LLM-augmented static analysis. With the rapid evolution of large language models, recent research has increasingly investigated their use in augmenting security-oriented program analysis, including vulnerability detection, binary analysis, reverse engineering, and automated patching [34, 66–69]. PatchAgent [69] leverages LLMs across multiple stages of the patching pipeline, including fault localization, patch generation, and validation, to reason about code semantics and support automated patching. DeGPT [66] leverages an iterative LLM-guided refinement loop to correct inaccuracies in decompiler outputs. Vul-BinLLM [67] applies in-context learning and reasoning-oriented prompts to analyze binaries, whereas LATTE [34] combines program slicing with LLM assistance to support binary-level taint analysis. Together, these works demonstrate that LLMs can support program-semantic understanding and reasoning over complex control-flow structures, which motivates our use of LLM-assisted analysis for LFVuln patch semantics.

10 Conclusion

In this paper, we present the first systematic study of incomplete LFVuln patches and develop *LinkDiff*, a binary differential analysis framework for detecting them. By analyzing 60 historical patches, we derive a fine-grained taxonomy of nine incompleteness patterns and incorporate these empirical insights into *LinkDiff* through retrieval-augmented LLM reasoning. On 60 historical incomplete patches, *LinkDiff* achieves 94.23% precision and 85.96% recall. Applying it to 76 recently released LFVuln patches uncovers 56 previously unknown incomplete patches, leading to 19 CVEs and $28,100 in bug bounties. These results demonstrate that incomplete remediation remains prevalent and security-critical, highlighting the need to treat patch completeness as a first-class concern in securing file-system software. We hope our findings and tooling will encourage both researchers and practitioners to treat patch completeness as a first-class concern when securing file-system software.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U2436207) and the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Yuan Zhang is the corresponding author. This paper was edited for grammar and style using ChatGPT. All AI-assisted revisions were manually reviewed and verified by the authors.

References

[1] Bocheng Xiang, Yuan Zhang, Fengyu Liu, Hao Huang, and Zihan Lin. Pig in a poke: Automatically detecting and exploiting link following vulnerabilities in windows file operations. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 7019–7038, 2025.

[2] Yu-Tsung Lee, Hayawardh Vijayakumar, Zhiyun Qian, and Trent Jaeger. Static detection of filesystem vulnerabilities in android systems. *arXiv preprint arXiv:2407.11279*, 2024.

[3] Chendong Yu, Yang Xiao, Jie Lu, Yuekang Li, Yeting Li, Lian Li, Yifan Dong, Jian Wang, Jingyi Shi, Defang Bo, et al. File hijacking vulnerability: The elephant in the room. In *NDSS*, 2024.

[4] Beibei Zhao, Wenjie Feng, Qingli Guo, Yingli Sun, Fangming Gu, Bolun Zhang, Xiaorui Gong, and Hong Li. Favdisco: Modeling and discovering file access vulnerabilities. *ACM Transactions on Software Engineering and Methodology*, 35(4):1–33, 2026.

[5] Can Huang, Xinhui Han, and Guorui Yu. Lpet–mining ms-windows software privilege escalation vulnerabilities by monitoring interactive behavior. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2089–2091, 2020.

[6] Sigmund Albert Gorski III, Seaver Thorn, William Enck, and Haining Chen. {FReD}: Identifying file {Re-Delegation} in android system services. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1525–1542, 2022.

[7] Suresh Chari, Shai Halevi, and Wietse Z Venema. Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS*, 2010.

[8] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. {JIGSAW}: Protecting resource access by inferring programmer expectations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 973–988, 2014.

[9] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. {RaceGuard}: Kernel protection from temporary file race vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[10] OpenAI. Agents. https://platform.openai.com/docs/guides/agents, 2026.

[11] Muhammad Arslan, Hussam Ghanem, Saba Munawar, and Christophe Cruz. A survey on rag with llms. *Procedia computer science*, 246:3781–3790, 2024.

[12] Zirui Song, Bin Yan, Yuhan Liu, Miao Fang, Mingzhe Li, Rui Yan, and Xiuying Chen. Injecting domain-specific knowledge into large language models: a comprehensive survey. *arXiv preprint arXiv:2502.10708*, 2025.

[13] Microsoft Corporation. Reparse points. https://learn.microsoft.com/en-us/windows/win32/fileio/reparse-points, 2025. Accessed: 2025-02-08.

[14] OWASP Foundation. Denial of service (dos). https://owasp.org/www-community/attacks/Denial_of_Service, 2025. Accessed: 2025-02-08.

[15] Microsoft Corporation. Windows error reporting. https://learn.microsoft.com/en-us/windows/win32/wer/windows-error-reporting, 2025. Accessed: 2025-02-08.

[16] Xuefeng Li and Zhiniang Peng. Exploiting errors in windows error reporting service in 2022. In *Power of Community (PoC) Conference*, 2022.

[17] Malwarebytes Corporation. Adwcleaner. https://www.malwarebytes.com/es/adwcleaner, 2025. Accessed: 2025-02-08.

[18] Microsoft Corporation. Getfileattributesw function. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getfileattributesw, 2025.

[19] Microsoft Corporation. File attribute constants: File_attribute_reparse_point. https://learn.microsoft.com/en-us/windows/win32/fileio/file-attribute-constants, 2025. Accessed: 2025-02-08.

[20] Microsoft Corporation. Deletefile function. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-deletefilew, 2025. Accessed: 2025-02-08.

[21] Razvan Raducu, Ricardo J Rodríguez, and Pedro Álvarez. Defense and attack techniques against file-based toctou vulnerabilities: A systematic review. *IEEE Access*, 10:21742–21758, 2022.

[22] Microsoft Corporation. Opportunistic locks. https://learn.microsoft.com/en-us/windows/win32/fileio/opportunistic-locks, 2025. Accessed: 2025-02-08.

[23] Zhiniang Peng and Qi Zhang. Diving into spooler: Discovering lpe and rce vulnerabilities in windows printer. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Diving-Into-Spooler-Discovering-Lpe-And-Rce-Vulnerabilities-In-Windows-Printer.pdf, 2021. Black Hat USA 2021.

[24] Tze Bin Bryan Pak. What's under the hood? root cause and patch analyses of elevation of privilege vulnerabilities in the windows operating system. In *IRC Conference on Science, Engineering and Technology*, pages 168–180. Springer, 2024.

[25] Tom Ganz, Erik Imgrund, Martin Härterich, and Konrad Rieck. Pavudi: Patch-based vulnerability discovery using machine learning. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 704–717, 2023.

[26] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3282–3299, 2021.

[27] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[28] Google, Inc. Bindiff: A tool for binary comparison. https://github.com/google/bindiff, 2025. Accessed: 2025-02-08.

[29] Hex-Rays. Ida pro: Interactive disassembler. https://www.hex-rays.com/products/ida/, 2025. Accessed: 2025-02-08.

[30] National Security Agency. Ghidra. https://github.com/NationalSecurityAgency/ghidra, 2019. Accessed: 2026-04-14.

[31] Grady Booch. Object-oriented design. *ACM SIGAda Ada Letters*, 1(3):64–76, 1982.

[32] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 195–211, 2019.

[33] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141, 2017.

[34] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, et al. Llm-powered static binary taint analysis. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–36, 2025.

[35] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Assisting static analysis with large language models: A chatgpt experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2107–2111, 2023.

[36] Linxi Jiang, Xin Jin, and Zhiqiang Lin. Beyond classification: Inferring function names in stripped binaries via domain adapted llms. In *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[37] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. *Advances in neural information processing systems*, 29, 2016.

[38] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[39] National Institute of Standards and Technology. National vulnerability database (nvd). https://nvd.nist.gov/, 2025. Accessed: 2025-02-08.

[40] Ross Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2010.

[41] CVE Program. CVE-2024-9244. https://www.cve.org/CVERecord?id=CVE-2024-9244, 2024. Accessed: 2026-07-21.

[42] Microsoft Security Response Center. Cve-2024-49107. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-49107, 2024. Accessed: 2025-02-11.

[43] Microsoft Security Response Center. Redirection guard: Mitigating unsafe junction traversal in windows. https://www.microsoft.com/en-us/msrc/blog/2025/06/redirectionguard-mitigating-unsafe-junction-traversal-in-windows, 2025.

[44] Microsoft Security Response Center. Cve-2025-54116. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2025-54116, 2025. Accessed: 2025-02-11.

[45] Microsoft Security Response Center. Cve-2020-1048. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2020-1048, 2025. Accessed: 2025-02-11.

[46] Microsoft. CreateFileW function. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew, 2024. Accessed: 2025-12-09.

[47] Microsoft. NT Path Namespace. https://learn.microsoft.com/en-us/windows/win32/fileio/naming-a-file, 2024. Accessed: 2025-12-09.

[48] Microsoft. UNC Path. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-dtyp/62e862f4-2a51-452e-8eeb-dc4ff5ee33cc, 2024.

[49] Microsoft. SetProcessMitigationPolicy. https://microsoft.github.io/windows-docs-rs/doc/windows/Win32/System/Threading/fn.SetProcessMitigationPolicy.html, 2024. Accessed: 2025-12-09.

[50] Microsoft. GetFinalPathNameByHandleW. https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getfinalpathnamebyhandlew, 2024.

[51] Microsoft Security Response Center. CVE-2020-1337. https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2020-1337, 2020. Accessed: 2025-12-09.

[52] Microsoft Security Response Center. CVE-2020-17001. https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2020-17001, 2020.

[53] CVE Program. Cve-2024-43114. https://www.cve.org/CVERecord?id=CVE-2024-43114, 2024. Accessed: December 15, 2025.

[54] Microsoft Security Response Center. CVE-2020-17014. https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2020-17014, 2020.

[55] Microsoft. Gettempfilename. https://learn.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-gettempfilename, 2024. Accessed: 2025-03-12.

[56] Microsoft. Impersonation. https://learn.microsoft.com/en-us/windows/win32/com/impersonation, 2020. Accessed: 2025-03-12.

[57] Ira Ray Jenkins, Sergey Bratus, Sean Smith, and Maxwell Koo. Reinventing the privilege drop: How principled preservation of programmer intent would prevent security bugs. In *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, pages 1–9, 2018.

[58] CVE Program. CVE-2023-36047, 2024. Accessed: 2026-07-21.

[59] Microsoft Security Response Center. Cve-2024-21447. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2024-21447, 2024. Accessed: 2025-02-11.

[60] Microsoft Security Response Center. Kb5053594. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2025-24983, 2025. Accessed: December 15, 2025.

[61] Microsoft. Dism operating system package servicing command-line options. https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/dism-operating-system-package-servicing-command-line-options?view=windows-11, 2024. Accessed: December 15, 2025.

[62] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[63] Dong-uk Kim, Junyoung Park, Sanghak Oh, Hyoungshick Kim, and Insu Yun. Windows plays jenga: Uncovering design weaknesses in windows file system security. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 3900–3914, 2025.

[64] Aditya Basu, John Sampson, Zhiyun Qian, and Trent Jaeger. Unsafe at any copy: Name collisions from mixing case sensitivities. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 183–198, 2023.

[65] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, Giuseppe Petracca, and Trent Jaeger. {PolyScope}:{Multi-Policy} access control analysis to compute authorized attack operations in android systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2579–2596, 2021.

[66] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium*, volume 267622140, 2024.

[67] Nasir Hussain, Haohan Chen, Chanh Tran, Philip Huang, Zhuohao Li, Pravir Chugh, William Chen, Ashish Kundu, and Yuan Tian. Vulbinllm: Llm-powered vulnerability detection for stripped binaries. *arXiv preprint arXiv:2505.22010*, 2025.

[68] Xinyu Hu, Zhiwei Fu, Shaocong Xie, Steven HH Ding, and Philippe Charland. Sok: Potentials and challenges of large language models for reverse engineering. *arXiv preprint arXiv:2509.21821*, 2025.

[69] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. Patchagent: A practical program repair agent mimicking human expertise. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security'25), Seattle, WA, USA*, 2025.

A Ethical Considerations

All experiments were conducted in a controlled environment without interacting with production systems or accessing real-world user data. Identified vulnerabilities were responsibly disclosed and remediated with affected vendors, and sensitive details were sanitized to prevent misuse.

B Open Science

To support transparency, we release the source code of *LinkDiff*, ten curated historical LFVuln patches with their analysis results, and detailed build and usage instructions. All artifacts are available at https://zenodo.org/records/20605695.

C Prompt Template

As shown in Figure 4, the prompt in *LinkDiff* encapsulates its three main components and leverages the LLM's reasoning capabilities to interpret the execution semantics of differential code fragments.

Patch Analyze Prompt Template

You are a reverse-engineering expert analyzing Link Following Vulns patch. Below is your task:

Goal: Locate the exact patch location in the repaired binary. Explain the patch logic and how it mitigates.
INPUT#1<Code Segment**>**: Differential code fragments appearing along the file-operation call chain
INPUT#2<Call Chain**>**: Corresponding file-operation call chain associated with modified code
CoT: (1) From the input call chain, identify the functions that contain modified code (f1, f2, ...), and first reason about their roles in the overall file-operation workflow.
(2) When encountering indirect calls, recall common analysis patterns for recovering real targets and use them to infer the concrete callees.
- Vtable Resolution: ...<Specific Analysis Process>...
- Function-Pointer Dereferencing: ...<Specific Analysis Process>...
- ...

(3) Whenever additional evidence is needed, invoke the functions in the Function Call.
(4) If the inferred patch behavior matches or closely resembles any ***Common Patch Repair Pattern***, or a similar remediation pattern, explicitly report the matched pattern.

Here is some information for **reference**:
<1> *Functions Call List*: You have the following functions for differential-database querying and for analyzing the binary in IDA.
...
<2> *Common Patch Repair Patterns*: Below are reference remediation patterns commonly used in link-following vulnerability fixes:
...
Output: A report with <**Summary**>, <**Patched Location**>, and <**Mitigation Analysis**>.

Figure 4: The prompt template in the *LinkDiff*

Table 5: Function calls exposed to the agent in *LinkDiff*

No.	Function Name	Description
1	get_metadata	Retrieve binary metadata.
2	get_function_by_name	Lookup function by name.
3	get_function_by_address	Lookup function by address.
4	list_functions	List all functions.
5	list_imports	List import functions.
6	decompile_function	Get the decompiled function code.
7	disassemble_function	Get the disassembled function code.
8	get_xrefs_to	Cross-references to function.
9	get_callees	List function callees.
10	get_callers	List function callers.
11	bindiff_open_database	Open BinDiff database.
12	bindiff_get_function_diff	Retrieve function differences.
13	data_read_byte	Read one byte.
14	data_read_word	Read 16-bit word.
15	data_read_dword	Read 32-bit dword.
16	data_read_qword	Read 64-bit qword.
17	data_read_string	Read string.
18	calculate_address	Compute absolute address.
19	calculate_relative_address	Compute relative address offset.

D Function Call List

The function calls in Table 5 fall into three categories: **(1) Functions 1–10** reconstruct execution paths and infer high-level semantics along file-operation call chains; **(2) Functions 11–12** retrieve patch differences and metadata for locating and interpreting modifications; and **(3) Functions 13–19** expose raw binary data, such as pointers, object layouts, and vtables, to resolve indirect control-flow transfers in LFVuln patch analysis.

E Missing Path Normalization

Listing 6 shows the patch for `CVE-2020-1337`, which detects path redirection by comparing the input path with the path resolved by *GetFinalPathNameByHandleW*. However, for handles opened through UNC paths, the API may return the original UNC string rather than a canonical path, allowing the check to be bypassed.

```
Handle h = CreateFile(path, GENERIC_ALL, ...);
// File handle path verification
if(IsPortALink(h, path)){
    vulnerable_file_operations();
}
HRESULT IsPortALink(HANDLE h, LPCWSTR Path){
    wchar_t szFilePath[520];
    // ... omitted initialization ...
    if (GetFinalPathNameByHandleW(h, szFilePath,...)){
        return S_FALSE; // Retrieve resolved path failed
    } else {
        if (_wcsnicmp(szFilePath, Path, PathLength)){
            return S_FALSE; // Redirection detected
        } else {
            return S_OK;
        }
}
```

Listing 6: Patch snippet for CVE-2020-1337.

F Missing Exception Handling

Listing 7 shows the patch for `CVE-2024-43114`, which creates the target directory and then applies restrictive access controls. However, missing error handling in this enforcement path introduces a new vulnerability, later assigned `CVE-2025-54530`.

```
/* Fix in the installation of TeamCity */
wchar_t* folderName = L"C:\\ProgramData\\JetBrains\\
     TeamCity\\system";
+++ if (CreateDirectoryW(folderName, NULL)){
    +++ SetSecurityFile(folderName, DACL_PROTECTED);
+++ }
/* Delete all files in folder */
WIN32_FIND_DATAW findFile;
HANDLE hFind = FindFirstFileW(folderName, &findFile);
while (FindNextFileW(hFind, &findFile)){
    DeleteFileW(findFile.cFileName);
}
```

Listing 7: Code snippet of the patch of CVE-2024-43114